AFRL-IF-RS-TR-2004-19
Final Technical Report
January 2004

COSAK: CODE SECURITY ANALYSIS KIT

Drexel University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. M140

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-19 has been reviewed and is approved for publication.

APPROVED: /s/
ELIZABETH S. KEAN
Project Engineer

FOR THE DIRECTOR: /s/
JAMES W. CUSACK, Chief
Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	JANUARY 2004	**FINAL** **Jun 00 – Dec 03**

4. TITLE AND SUBTITLE

COSAK: CODE SECURITY ANALYSIS KIT

5. FUNDING NUMBERS

G - F30602-01-2-0534
PE - 62301E
PR - CHAT
TA - 00
WU - 03

6. AUTHOR(S)

Spiros Mancoridis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Drexel University
Department of Computer Science, COE
3141 Chestnut Street
Philadelphia PA 19104

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFSA
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2004-19

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Elizabeth S. Kean/IFSA/(315) 330-2601 Elizabeth.Kean@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

There are two significant parts to the Code Security Analysis Kit (CoSAK) project. The first part of the project is called Front Line Functions (FLF) and involves the development of static analysis tools for C code to assist in the characterization of software functions that are most vulnerable to a security attack. The effectiveness of the FLF work was demonstrated empirically using a repository of open source software with known security vulnerabilities. The second part of the project is called Gemini and involves the development of tools to transform C programs into equivalent ones that are less susceptible to a buffer overflow security attack. The effectiveness of the Gemini project was demonstrated using a case study that involved transforming several software packages from the Linux operating system distribution.

14. SUBJECT TERMS

Code Security, Static Analysis, Security Vulnerability Analysis

15. NUMBER OF PAGES 17

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

Methods, Assumptions, and Procedures

The classic scientific method was used to develop the FLF tools. The data for our experiment was compiled as a repository of 30 open source software systems with known security vulnerabilities. Each software system in the repository contains exactly one security flaw, which is documented as a source code patch file. Since source code patches typically repair several bugs, it was difficult to find patch files that repaired only one security flaw. The hypothesis that we wanted to test was that security vulnerabilities typically involve functions that perform input/output operations or are close to ones that perform input/output operations. Examples of input/output operations include the opening, reading, and writing of file or socket data.

Our hypothesis was shown to be true for all 30 software systems that were in the repository. The next step was to develop a tool, called FLFfinder that can automatically find all functions that either perform or are close to the functions that perform input/output operations. The effectiveness of this tool was verified on three open source software systems that have known vulnerabilities but were not part of our experimental data.

A case study was used to validate the effectiveness of the Gemini tool. Specifically, several open source software systems were transformed into semantically equivalent program that were not as susceptible to buffer overflow security attacks. After transforming the source code of each system we verified that the system compiled and passed its regression test suite correctly. We also designed examples of software with exploitable buffer overflow security vulnerabilities and showed that the transformations performed by the Gemini tool mitigated them.

Results and Discussion

FLF Finder Tool

A software vulnerability is a fault in the specification, implementation, or configuration of a software system whose execution can violate an explicit or implicit security policy. Software maintainers typically focus on the functionality of software rather than on its security posture. Hence, vulnerabilities often escape their attention until the software is exploited by specially written malicious code.

A large percentage of software is developed using unsafe programming languages (e.g., C and C++) in the name of cost effectiveness, programmer

familiarity, and performance. Being unable to influence how others develop new software, we must find ways to improve the maintenance process to secure software against possible attacks.

Code audits are one aspect of the maintenance process that can expose security vulnerabilities. Audits have been tried, with some success, on systems such as the OpenBSD operating system. Unfortunately, audits are expensive and reoccurring. Each audit requires many man-hours, and each software revision requires re-examination to verify that new faults have not been introduced.

The quantity of code in many systems makes large-scale auditing infeasible. In the case of OpenBSD, the auditing effort only focuses on software that is enabled in the default installation. This decision has resulted in overlooked vulnerabilities in often-used components of the distribution that have not been audited, such as `telnetd`.

Beizer states that good source code will have one to three faults for every one hundred lines of code. However, it is not known which of those faults is a security fault. Auditors would benefit from a tool that can reduce the amount of code that needs to be studied; enabling them to focus their attention on areas of likely vulnerability.

Our hypothesis is that a small percentage of functions near a source of input (e.g., file I/O) are the most likely to contain a security fault. We refer to these functions as FLFs (Front Line Functions), and the percentage of functions likely to contain a security fault as the FLF density. We validate our hypothesis with an experiment that involves 31 open source systems using two tools that we developed for this purpose.

Based on the validation of the hypothesis, the FLF Finder tool was developed to identify areas of high vulnerability likelihood automatically. The effectiveness of the FLF Finder is demonstrated in two ways. First, it is applied to three open source software systems, micq, elm, and dhcpd, each with known (documented) vulnerabilities. Second, the FLF Finder is applied to the OpenSSH server daemon, which does not have known vulnerabilities but has recently undergone a widely publicized restructuring, called privilege separation. This separation aims at minimizing the amount of code that runs with elevated privileges. By minimizing the amount of privileged code, it reduced the risk of a security vulnerability occurring within that code. Although the restructuring was done manually, our case study shows that the results produced by the FLF Finder are consistent with the design decisions made by the maintainers.

For the purpose of the FLF experiment, we refer to the functions that accept input as Inputs and the functions with known vulnerabilities as Targets. An example of an Input is a user-defined function that contains a call to `read`, defined in `unistd.h`. The function that invokes `read` stores data from a possibly untrusted source in a buffer. Our analysis revealed that the most common sources of input are user supplied input, input via command line arguments, and input from environment variables.

The FLF Finder supplies a list of common Inputs such as `read`. However, any function could be a potential Input, so the user may specify what other functions in the software system are considered Inputs.

A Target is any function that contains a known vulnerability. These functions typically use a global buffer or a variable parameter that contains data from an Input. For example, a Target could be a function that calls `printf` using user-supplied input as the first argument. If nothing is known about the software system then all functions in the system are Potential Targets.

All of the open source systems used in the experiment have at least one known security fault and a patch file for repairing the vulnerability. Maintainers using the Unix diff tool create the patch files. The experiment uses patch files to identify the Targets in each system automatically.

The experiment uses two tools that we developed. The GNU Abstract Syntax Tree Manipulation Program (GAST-MP) takes pre-processed C source code and generates a database of code facts for each system in the experiment. The System Graph Analyzer (SGA) discovers Targets in the source code and creates the necessary function call graphs.

The GNU C++ Compiler (G++) can output the abstract syntax tree (AST) as an ASCII text file when given the `-fdump-tree-original flag`. GAST-MP parses this file and produces a relational database of code facts.

The SGA tool has a dual purpose. It functions as a vulnerability patch file analyzer to identify Target functions and as a function call graph generator that is used to trace how potentially dangerous data could flow from Inputs to Targets.

For each vulnerability patch file, SGA determines the line number of each subtractive line in the corresponding source code. It then uses the GAST-MP database to find the function that contains that line of code. Once the function is determined, it is marked as a Target.

Recall that the FLF hypothesis states that a small percentage of functions, specifically those near a source of input, are most likely to contain a security vulnerability. We will show that in 31 open source systems FLFs occur within close proximity to an Input. The proximity is measured as the number of function invocations that occur between the Input and Target.

The FLF density k represents the percentage of Potential Targets in a software system that transmits data from an Input. Thus k can be computed via the ratio p/m. p is the number of functions involved in a function invocation path between an Input and Target; m represents the total number of Potential Targets.

The validation of the FLF hypothesis consists of four stages. The first stage is to search for software systems with known vulnerabilities and patch files for those vulnerabilities. In general, it is difficult to find patch files that only pertain to security vulnerabilities since maintainers often make one general patch file that contains fixes for both regular faults and security vulnerabilities. Fortunately, some Linux distributions provide software in the form of Source Red Hat Package Manager (SRPM) files. SRPMs contain unaltered source code

and a set of patches that address specific faults in the source. SRPM packages comprise much of the test suite.

The second stage is to pre-process each software system in the test suite with G++ to resolve macros and compile-time dependencies. GAST-MP is then used to generate a database of code facts for each system.

Finally, SGA is used to calculate the FLF density of each system. The process to calculate FLF density is detailed in our paper [ICSM03].

Our experiment computed the FLF density

for each system. The sample mean FLF density across all systems is 2.87% with a standard deviation of 1.83. This means that, on average 2.87% of the functions in each system were involved in the security vulnerability documented by the patch files.

The FLF Finder discovers those functions in the code that are at high risk of vulnerability. The tool is not intended to find faults, only to show which functions are at risk. The FLF Finder requires two pieces of information to be provided by the user. The first is the source code to be analyzed, and the second is a list of Inputs (besides those provided by the FLF Finder.

The process the FLF Finder uses is the following algorithm:

1. Create the entire call graph G for the system.
2. Label Input nodes in G=(V,E) with a depth of 0.
3. Compute the total number of functions, m, in G.
4. Given the FLF density result of 2.87%, solve for p <- km
5. Label the nodes in G as follows:

- Perform a reverse (follow incoming edges) breadth first search (bfs) from each 0-labeled node to depth p. During the reverse bfs, if a visited node is not already labeled, label it with its depth (e.g., 1,2,3, …).
- For all labeled nodes u in set V, perform a bfs from its labeled depth ending at depth p. All nodes visited are at high risk of vulnerability.

This process is identical to the process used to compute the FLF density except that it might produce false positives. The false positives are introduced because Targets are not known ahead of time. In the experiment, the resulting FLF density was based on one path through one common ancestor. Since the Targets are not known ahead of time when the FLF Finder is used, every function invocation path of length p through every common ancestor is suspect. Step 5 of the algorithm is responsible for finding all three invocation paths discussed in previously.

To test the effectiveness of the FLF Finder, we applied it to three open source systems, micq, elm, and dhcpd, with known vulnerabilities that were not used in the experiment.

Only dhcpd failed to identify all of its known vulnerabilities. It failed to find one vulnerability because there is no known path to the function in dhcpd which contains the vulnerability.

One of our objectives was to supply the maintainer with a tool that eases the process of performing a security audit. The FLF Finder accomplishes this by eliminating most of the system's functions from consideration.

We applied the FLF Finder to a software system with no known vulnerabilities. In the previous section, patch files had been used to test accuracy. In a system with no patch files we use the maintainers' design decisions to evaluate the success of the FLF Finder.

OpenSSH is a free suite of network connectivity tools that a growing portion of the Internet is relying on. Telnet, rlogin, ftp, and other such programs transmit unencrypted password information during authentication. OpenSSH encrypts all traffic (including passwords) to eliminate eavesdropping, connection hijacking, and other network-level attacks. OpenSSH includes sshd, a secure alternative to telnetd, and sftp, a secure alternative to ftp.

The principle behind privilege separation is to minimize the amount of code that runs with elevated privileges without limiting the functionality of the program. Privilege, in this context, refers to a "security attribute that is required for certain operations". The result of privilege separation is that the separated code, which runs with elevated privileges, can now be audited thoroughly due to its small size. After separation, the number of lines of sshd code that needed to be audited was reduced from approximately 20,000 to 2,000.

In OpenSSH, privilege separation is implemented via a message passing API. The API is used to transmit data between an unprivileged section and a privileged section, and vice versa. The actual details of the message passing scheme involve interprocess communication (IPC) and are beyond the scope of this report.

The unprivileged code executes as a slave. A slave is an unprivileged user that is sandboxed in a specific directory. Any attempts to exploit an application (i.e., OpenSSH) should result in either a denial of service to the attacker or the execution of arbitrary instructions as the slave.

Privilege separation was originally an optional part of the OpenSSH architecture. However, the OpenSSH team made it mandatory as of version 3.2.3.

The goal of the study is to demonstrate that the FLF concept and the FLF Finder can be used to increase the efficiency of a source code auditor by identifying functions that are at high-risk of containing a security fault. Comparing the FLF Finder results of OpenSSH 3.1 against OpenSSH 3.2.3 will do this. We will show that the functions identified by the FLF Finder correspond to the functions modified by the maintainers in 3.2.3 while implementing privilege separation. The process that we follow can be best explained with set notation:

- F_3.1 - The set of functions found by the FLF Finder.
- P_3.2.3 – The set of functions that are involved in privilege separation.
- A_3.1 - The set of all functions in 3.1.
- C - The intersection of P_3.2.3 and F_3.1.
- S - The intersection of I and A_3.1.

We first run the FLF Finder on 3.1; this results in F_3.1. F_3.1 includes 34% (i.e., 374) of the functions in 3.1. the functions that are involved in privilege separation.

The set P_3.2.3 also contains functions that are not found in 3.1. The functions in P_3.2.3 that are not in 3.1 must be removed in order to consider only the functions in 3.1 that were changed in 3.2.3. The set C contains the functions in 3.1 that were modified while implementing privilege separation in 3.2.3. The cardinality of C is 17. To verify the success of the study, we intersect F_3.1 with C.

We found that of the 374 functions contained in F_3.1, the FLF Finder identified 14 of the 17 functions involved in privilege separation, resulting in an accuracy of 82%. Therefore, by reviewing 34% of OpenSSH, we were able to identify 82% of the functions that were modified to increase OpenSSH's security posture in 3.2.3.

This case study presents an example of how the FLF concepts and tools can be used to aid code auditors in finding high-risk areas of code in an efficient manner. We were able to remove at least 10,000 lines of code from consideration with very little effort while maintaining a high degree of accuracy. As our tools mature and our experimental set become larger we hope that our ability to reduce the unimportant segments of a system (in terms of security) will improve.

Gemini Tool

Buffer overflows are the most common source of security vulnerabilities in C programs. This class of vulnerability, which is found in both legacy and modern software, costs the software industry hundreds of millions of dollars per year.

The most common type of buffer overflow is the run-time stack overflow. It is common because programmers often use stack allocated arrays. This enables the attacker to change a program's control flow by writing beyond the boundary of an array onto a return address on the run-time stack. If the arrays are repositioned to the heap at compile time, none of these attacks succeed. Furthermore, repositioning buffers to the heap should perturb the heap memory enough to prevent many heap overflows as well.

We have created a tool called Gemini that repositions stack allocated arrays at compile time using TXL. The transformation preserves the semantics of the program with a small performance penalty. Our approach involves the semantics-preserving transformation of stack allocated arrays to heap allocated “pointers to arrays'”. A program that is amenable to a buffer overflow attack and several Linux programs were used as examples to demonstrate the effectiveness and overhead of our technique.

C is a widely used programming language for critical software (e.g., operating systems and system software). Most of the software that is bundled with Linux and Sun Solaris are written in C. Furthermore; the most popular servers on the Internet for e-mail, the World Wide Web, and the Domain Name System are implemented in C.

C programmers often use arrays to store data gathered from external input. Stack allocated arrays are automatic variables; hence they are allocated and de-allocated during run-time without programmer intervention. This is convenient since the input is often used immediately (see previous discussion of FLFs). Despite their convenience, stack allocated arrays are vulnerable to buffer overflow attacks. Fortunately, allocating all arrays to the heap can mitigate such attacks.

Stack buffer overflows are the most common form of security vulnerability found in C programs. This vulnerability alone costs industry hundreds of millions of dollars per year. For example, `bind`, the software responsible for 95% of the Domain Name System, was discovered to contain a buffer overflow as recently as November, 2002. After the discovery of vulnerabilities in infrastructure-critical software, many man-hours of software analysis, reinstallation, and testing are required to fix it.

Moving stack allocated arrays to the heap accomplishes two things. First, it disrupts the attack vectors of known stack buffer overflow exploits and all future stack buffer overflow exploits. Second, it can disturb the heap memory enough to eliminate known heap buffer overflow attack vectors also. Moving a stack allocated array to the heap does not fix the bug that causes the buffer overflow, it only prevents the overflow from providing the attacker with elevated privileges, such as a command shell. This leads to less vulnerability in the long run since it is very difficult, and in many cases impossible, for an attacker to leverage a heap buffer overflow.

In C, a heap allocated buffer is actually a pointer to contiguous memory. Pointers are not automatic variables; hence they require explicit memory management by the programmer. The added complication of explicit memory management often leads to bugs such as uninitialized pointers and memory leaks. A program that transforms arrays into “pointers to arrays” can automate memory management. Such a program should preserve the semantics of the original program so that the transformation is transparent. In C, this is a problem since arrays and pointers are not equivalent types.

Preserving the semantics of the program after the transformation allows code to be developed using conventional programming practices (i.e., allocating certain buffers on the stack). Furthermore, maintenance and debugging need not be hampered by the prolific use of pointers. Rather, the code is automatically transformed to use heap allocated “pointers to arrays” immediately prior to compilation.

We have created a tool called Gemini that uses TXL rules to transform stack allocated arrays into heap allocated “pointers to arrays” automatically. This transformation preserves the semantics of the original program, allowing it to be inserted into the end of the development process transparently and with a small amount of run-time overhead.

Our work is related to two major areas of research. The first is software security, specifically as it applies to buffer overflow vulnerabilities in code. The second is the use of source code transformation for code re-engineering.

Buffer overflows may occur when a fixed size memory allocation is used to store a variable-size data entry. There are conflicts when the variable-size data entry overruns the bounds of the fixed-size memory. These overflows are typically exploited by entering a string that is larger than the buffer assigned to hold it. If the return address (RA) is part of the overwritten run-time stack, an attacker may execute arbitrary code, such as spawning a remote terminal session.

Unlike the stack, the heap does not contain return addresses, making it harder to change the program's control flow.

Some security tools, such as Splint, perform static analysis to find code that is likely to be vulnerable. Unlike our technique, however, they require programmers to annotate their source code with constraints. Not all of the existing source code analysis tools require code annotations, however.

StackGuard has been reasonably successful at reporting buffer overflows immediately after they happen at run-time. Specifically, StackGuard inserts code into the application at compile time and a 'canary' value just before the return addresses on the run-time stack. When the function returns, the added code checks if this canary value is still in place. If the canary value is no longer present, a buffer overflow must have occurred. When this happens, the application terminates with a notification.

A way to avoid the side effects of an exploited vulnerability is to disallow the execution of the run-time stack. This prevents executable code, such as shell instructions that may have been placed on the stack during a buffer overflow, from being executed.

One way to get around a non-executable run-time stack is to perform a heap overflow, followed by a stack overflow. The heap overflow is used to insert the binary instructions for a command shell into the program's executable memory space. A stack overflow is then used to modify the return address of the current stack frame to point to the executable shell instructions in the heap.

Several languages have been created to perform source code transformation. One such language is TXL.

TXL uses a grammar for the input text to be transformed and a set of rules for performing the transformations. TXL can be thought of as a mixture of a functional programming language and the Unix tools like `lex` and `yacc`. The TXL grammar files are specified in extended Backus-Naur form. First, TXL uses the specified grammar files to produce a scanner and parser for that grammar. Second, it generates a parse tree from the input using the scanner and parser. Finally, it applies the transformation rules to the tree.

In order to perform the transformation, one simplifying assumption is made. The C source code being transformed must be compilable to an executable binary. Recall that we are attempting to prevent stack allocated buffer overflows. This is accomplished by transforming all stack allocated arrays into heap allocated "pointers to arrays". This transformation preserves the semantics of array access and function argument declarations.

Array declarations within functions are the only arrays that must be transformed. Arrays that are declared outside the scope of a function are

allocated in the block storage segment and data segment of the executable, and hence, they are not vulnerable to stack overflows. Similarly, pointers to arrays and pointers to pointers, allocated with `malloc`, are on the heap and therefore not vulnerable to stack-based buffer overflow attacks.

The following steps outline our transformation process.

1. Declaration Expansion. This step expands declarations that contain a list of declarators. Expanding declarations simplifies the rest of the transformations.
2. `typedef` Flattening. A `typedef` can either alias a type or an array of types. Without doing the flattening, there could be many nested `typedef` aliases, making it difficult to determine the correct "pointer to array" declaration.
3. Declaration Transformation. This step transforms all local array declarations to "pointer to array" declarations. An initialization function is created to perform all memory allocation and initialization for the pointer to array. The memory allocation and initialization cannot be performed within the body of the function due to an ambiguity in the C grammar concerning declarations and statements. Due to this ambiguity, it is impossible to guarantee that the allocation and initialization of each transformed array will occur before any statements reference the resulting pointer. To solve this problem, we perform all of the work in a separate function. This function returns a pointer to the prepared memory. The ISO C99 specification allows the dimensions of locally defined arrays to be variable sized. After the transformation, the dimensions of the resulting pointer will be referenced several times during initialization. Simply copying the expression to several areas of the source code would result in a failure to preserve the semantics of the program. To solve this problem, the dimensions of the original array are extracted and stored in a local integer variable. This placeholder is substituted for the original expression during allocation and initialization of the memory.
4. `sizeof` alias Declarations. This step inserts a new, unique array declaration for each array declaration that was transformed. This new declaration is the same as the original array declaration, except that it is not initialized with data. The purpose of the unique declaration is to preserve the semantics of `sizeof`. To be proper, the `sizeof` constant should only be passed a type. However, many programmers will pass it a variable or an expression. If a `sizeof` constant references the transformed array, it will no longer evaluate to the same value as the original program since the type has changed from array to pointer. In order to solve this special case, we search through the scope of each transformed array declaration and replace every reference to the original array, within a `sizeof`, with the name of the new unique declaration.
5. Add `free` and Transform `return` and `sizeof`. This step adds the appropriate calls to `free`, and transforms the `return` and `sizeof` statements. The calls to `free` are necessary to preserve the behavior of the original arrays, which are automatic variables. The transformation will insert

the `free` calls at the end of every block where the original array would have run out of scope. If the `return` statement references one of the buffers, a segmentation fault may occur since the `return` statement will attempt to dereference an invalid pointer. Hence, the expression that would have been returned is stored in a local variable, and the contents of the variable are returned instead.

6. Initialization Functions. This step adds the initialization functions to the end of the source file to ensure that any necessary header files are included above them, such as `stdlib.h`. Prototypes for the new functions are inserted at the top of the source file so that the compiler can resolve the symbol names of the functions.

To demonstrate the effectiveness of our transformation, we show how the transformation of source code that is amenable to a buffer overflow prevents the exploit from occurring. Several transformed Linux programs have been tested to demonstrate the expected efficiency of the transformed code.

To show the amount of overhead that can be expected from using heap allocated buffers in place of stack allocated arrays, we transformed several Linux programs, each with varying degrees of size and complexity. If a program came with a regression test suite, these tests were performed on both the original code and the transformed code. The binaries were compiled without optimizations in each case. The time increase was calculated in one of two ways. If the program did not include a suite of regression tests, it was executed fifty times with standard options. The result of this was compared to the same tests being executed on the non-transformed binary. If the program did include a suite of regression tests, the time increase was calculated by taking the difference in fifty runs of the test suite of the transformed and non-transformed binaries.

The following steps constitute the pipeline for transforming C code. This pipeline can be inserted directly into the build process of most open source software.

1. Configure and build the program. This ensures that all necessary build files are created and that the program holds the simplifying assumption mentioned previously. From this step we obtain the names of the files that need to be transformed.
2. Automatically modify the `Makefile`. Use a `sed` script to automatically change the `Makefile` so that it produces pre-processed C code instead of object files and binaries. Finally, update the modification time of each C file using the `touch` program. This ensures that `make` will attempt to generate object files and binaries in the next step.
3. Generate pre-processed C. Execute `make` again for each file that was produced during the initial execution of `make`. This time the pre-processed C code will be produced for each file. The debugging directives found in the pre-processor output from GCC are removed since the TXL grammar cannot parse them.

4. Transform the program. Backup the original C source files and transform each pre-processed C file, overwriting the original C source file with the transformed output. After all of the files have been transformed, execute `make` again to create the transformed program, then restore the original C source files.

Conclusions

The FLFinder and Gemini tools in CoSAK represent a 2-pronged approach to securing software systems.

The FLF hypothesis states that a relatively small percentage of functions near a source of input are the most likely to contain security vulnerabilities. We ran an experiment to validate this hypothesis. The results of this experiment support our hypothesis. These results were tested against several open source software systems not included in the experiment, as well as the OpenSSH server daemon. The case study showed that the design decisions made by the OpenSSH team concur with the results our FLF Finder produced. By using the FLF Finder, code auditors can focus their attention on the most vulnerable functions in the system. This would allow them to spend more of their time searching for less obvious security flaws in systems, leading to more secure applications.

The Gemini tool guarantees that current and future stack buffer overflow attack vectors will fail when used against a transformed program, since the heap does not contain return addresses.

Our solution does not fix the bug that causes a buffer overflow, but it does mitigate the risk of such a bug by preventing the attacker from inserting executable instructions, such as shell instructions, and overwriting the return address to jump to those instructions. Furthermore, our technique preserves the semantics of the program. This allows Gemini to be inserted into the regular development process effortlessly.

The performance associated with using heap memory instead of stack memory will increase with the amount of use the stack allocated buffers receive, and by the number of times the functions containing the arrays are called. A fortunate side effect of our technique is that by inserting more buffers onto the heap, the heap memory becomes perturbed. This perturbation lends itself to thwarting current and future heap buffer overflow attack vectors. Gemini is available from our website at http://serg.cs.drexel.edu/gemini/.

Recommendations

The FLF work can be viewed as a foundation for future work on developing more secure and fault tolerant software. Specifically, our long-term plan is to provide mechanisms that will allow code to continue running even in the presence of faults through isolation, policy relaxation and dynamic reconfiguration. These capabilities will reduce the likelihood of successful

denial of service attacks, and provide a better understanding of the software failure or the techniques used by the attacker.

We plan to achieve these goals by defining the operational envelope of a software system (i.e., the exact runtime requirements of each of its components), and then creating a special runtime environment that ensures that the program stays within its envelope. We also define an area outside of the normal envelope, which we call the *red zone*. If the program enters the red zone, we know that the behavior of the program is off nominal, but we do not know the cause and the severity of the problem. Rather than terminating the program, we place the runtime environment into an increased state of readiness and allow the code to continue running. We also take actions to limit the damage that the misbehaving program can cause. If the program attempts to breach the red zone, it will be terminated by the runtime system.

The envelope of the system is defined as the set of conditions that remain true throughout the lifetime of each of the system components. These conditions are essentially the operational parameters of each component. If any of them are violated, the component is operating outside of its design limits as a result of a software problem, or an attack. We refer to these conditions, along with the actions performed by the runtime system in response to the violated conditions, as *contracts*. If, during the execution of the program, a contract is violated, the runtime environment is provided with sufficient information about the program (via the contracts) to determine what kind of action should be taken.

Using the CoSAK tools we can determine which functions in a system are FLFs (i.e., Front Line Functions). Recall that an FLF is a function that is close to a source of input from the external environment (e.g., a function that reads from a file, socket, standard input). As it would be prohibitively expensive to specify contracts for all of the function in a software system, we can use heuristics such as the FLF concept to direct the attention of developers to the most vulnerable functions from a reliability and security perspective.

After the FLFs have been identified we will use our profiling tools to determine the nominal behavior of the application undergoing scrutiny. Subsequently, we will document this nominal behavior using contracts written the contract specification language. We will, finally, test these same systems under the red zone-enabled runtime system to check that the faults and security vulnerabilities no longer result in simple program terminations.

References

[WCRE03] Dahn, C. and Mancoridis, S. *Using Program Transformation to Secure Against Buffer Overflows*, In the IEEE Proceedings of the 2003 Working Conference in Reverse Engineering (WCRE'03), *Victoria*, BC, Canada, 2003.

[ICSM03] DaCosta, D. and Dahn, C. and Mancoridis, S. and Prevelakis, V. *Characterizing the Security Vulnerability Likelihood of Software Functions*, by In the IEEE Proceedings of the 2003 International Conference on Software Maintenance (ICSM'03), Amsterdam, The Netherlands, September, 2003.

Glossary

CoSAK. The Code Security Analysis Kit is a collection of software tools to help analyze C source code to (a) determine which of its functions are more amenable to as security attack and (b) transform C programs so that they are less susceptible to certain types of buffer overflow attacks.

FLF. Front Line Functions are the functions in a program that either perform input/output or that interact closely with functions that do.

FLFfinder. The FLFfinder tool analyzes C programs and finds the program's FLFs.

Gemini. The Gemini tool transforms C code into semantically equivalent C code that is not as susceptible to certain kinds of buffer overflow attacks.